Exhibit LL

PUBLIC VERSION

QUALCOMM®

Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ V66

Programmer's Reference Manual

80-N2040-42 A

November 17, 2017

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

1 Introduction

The Qualcomm Hexagon™ processor is a general-purpose digital signal processor designed for high performance and low power across a wide variety of multimedia and modem applications. V66 is a member of the sixth generation of the Hexagon processor architecture.

1.1 Features

- **Memory**

  Program code and data are stored in a unified 32-bit address space. The load/store architecture supports a complete set of addressing modes for both compiler code generation and DSP application programming.

- **Registers**

  Thirty two 32-bit general purpose registers can be accessed as single registers or as 64-bit register pairs. The general registers hold all data including scalar, pointer, and packed vector data.

- **Data types**

  Instructions can perform a wide variety of operations on fixed-point or floating-point data. The fixed-point operations support scalar and vector data in a variety of sizes. The floating-point operations support single-precision data.

- **Parallel execution**

  Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution, with each packet containing from one to four instructions. Vector instructions operate on single instruction multiple data (SIMD) vectors.

- **Program flow**

  Nestable zero-overhead hardware loops are supported. Conditional/unconditional jumps and subroutine calls support both PC-relative and register indirect addressing. Two program flow instructions can be grouped into one packet.

- **Instruction pipeline**

  Pipeline hazards are resolved by the hardware: instruction scheduling is not constrained by pipeline restrictions.

- **Code compression**

Compound instructions merge certain common operation sequences (add-accumulate, shift-add, etc.) into a single instruction. *Duplex* encodings express two parallel instructions in a single 32-bit word.

- **Cache memory**

  Memory accesses can be cached or uncached. Separate L1 instruction and data caches exist for program code and data. A unified L2 cache can be partly or wholly configured as tightly-coupled memory (TCM).

- **Virtual memory**

  Memory is addressed virtually, with virtual-to-physical memory mapping handled by a resident OS. Virtual memory supports the implementation of memory management and memory protection in a hardware-independent manner.

Figure 1-3 presents an overview of the instruction classes and how they can be grouped together.

Slot 0	Slot 1	Slot 2	Slot 3
LD Instructions ST Instructions ALU32 Instructions MEMOP Instructions NV Instructions SYSTEM Instructions Some J Instructions	LD Instructions ST Instructions ALU32 Instructions Some J Instructions	XTYPE Instructions ALU32 Instructions J Instructions JR Instructions	XTYPE Instructions ALU32 Instructions J Instructions CR Instructions

XTYPE Instructions (32/64 bit)
Arithmetic, Logical, Bit Manipulation
Multiply (Integer, Fractional, Complex)
Floating-point Operations
Permute / Vector Permute Operations
Predicate Operations
Shift / Shift with Add/Sub/Logical
Vector Byte ALU
Vector Halfword (ALU, Shift, Multiply)
Vector Word (ALU, Shift)

ALU32 Instructions
Arithmetic / Logical (32 bit)
Vector Halfword

CR Instructions
Control-Register Transfers
Hardware Loop Setup
Predicate Logicals & Reductions

NV Instructions
New-value Jumps
New-value Stores

J Instructions
Jump/Call PC-relative

JR Instructions
Jump/Call Register

LD Instructions
Loads (8/16/32/64 bit)
Deallocframe

ST Instructions
Stores (8/16/32/64 bit)
Allocframe

MEMOP Instructions
Operation on memory (8/16/32 bit)

SYSTEM Instructions
Prefetch
Cache Maintenance
Bus Operations

Figure 1-3 Instruction classes and combinations

1.3.11 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics which are used to directly express Hexagon processor instructions from within C code. For example:

```
int main()
{
    long long v1 = 0xFFFF0000FFFF0000;
    long long v2 = 0x0000FFFF0000FFFF;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions.

1.4 Notation

This section presents the notational conventions used in this document to describe Hexagon processor instructions:

- Instruction syntax
- Register operands
- Numeric operands

NOTE The notation described here does not appear in actual assembly language instructions. It is used only to specify the instruction syntax and behavior.

1.4.1 Instruction syntax

The following notation is used to describe the syntax of instructions:

- `Monospaced font is used for instructions`
- Square brackets enclose optional items (e.g., `[:sat]`, means that saturation is optional)
- Braces indicate a choice of items (e.g., `{Rs,#s16}`, means that either Rs or a signed 16-bit immediate can be used)

11.10 XTYPE

The XTYPE instruction class includes instructions which perform most of the data processing done by the Hexagon processor.

XTYPE instructions are executable on slot 2 or slot 3.

11.10.1 XTYPE/ALU

The XTYPE/ALU instruction subclass includes instructions which perform arithmetic and logical operations.

Absolute value doubleword

Take the absolute value of the 64-bit source register and place it in the destination register.

Syntax	Behavior
`Rdd=abs(Rss)`	`Rdd = ABS(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=abs(Rss)`	`Word64 Q6_P_abs_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	0	d	d	d	d	d	Rdd=abs(Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

11.10.2 XTYPE/BIT

The XTYPE/BIT instruction subclass includes instructions for bit manipulation.

Count leading

Count leading zeros (cl0) counts the number of consecutive zeros starting with the most significant bit.

Count leading ones (cl1) counts the number of consecutive ones starting with the most significant bit.

Count leading bits (clb) counts both leading ones and leading zeros and then selects the maximum.

The NORMAMT instruction returns the number of leading bits minus one.

For a two's-complement number, the number of leading zeros is zero for negative numbers. The number of leading ones is zero for positive numbers.

The number of leading bits can be used to judge the magnitude of the value.

Syntax	Behavior
`Rd=add(clb(Rs),#s6)`	`Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))+#s;`
`Rd=add(clb(Rss),#s6)`	`Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))+#s;`
`Rd=cl0(Rs)`	`Rd = count_leading_ones(~Rs);`
`Rd=cl0(Rss)`	`Rd = count_leading_ones(~Rss);`
`Rd=cl1(Rs)`	`Rd = count_leading_ones(Rs);`
`Rd=cl1(Rss)`	`Rd = count_leading_ones(Rss);`
`Rd=clb(Rs)`	`Rd = max(count_leading_ones(Rs),count_leading_ones(~Rs));`
`Rd=clb(Rss)`	`Rd = max(count_leading_ones(Rss),count_leading_ones(~Rss));`

`Rd=normamt(Rs)`	`if (Rs == 0) {`
	`    Rd = 0;`
	`} else {`
	`    Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))-1;`
	`}`

`Rd=normamt(Rss)`	`if (Rss == 0) {`
	`    Rd = 0;`
	`} else {`
	`    Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))-1;`
	`}`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=add(clb(Rs),#s6)`	`Word32 Q6_R_add_clb_RI(Word32 Rs, Word32 Is6)`
`Rd=add(clb(Rss),#s6)`	`Word32 Q6_R_add_clb_PI(Word64 Rss, Word32 Is6)`
`Rd=cl0(Rs)`	`Word32 Q6_R_cl0_R(Word32 Rs)`
`Rd=cl0(Rss)`	`Word32 Q6_R_cl0_P(Word64 Rss)`
`Rd=cl1(Rs)`	`Word32 Q6_R_cl1_R(Word32 Rs)`
`Rd=cl1(Rss)`	`Word32 Q6_R_cl1_P(Word64 Rss)`
`Rd=clb(Rs)`	`Word32 Q6_R_clb_R(Word32 Rs)`
`Rd=clb(Rss)`	`Word32 Q6_R_clb_P(Word64 Rss)`
`Rd=normamt(Rs)`	`Word32 Q6_R_normamt_R(Word32 Rs)`
`Rd=normamt(Rss)`	`Word32 Q6_R_normamt_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	0	0	0	d	d	d	d	d	Rd=clb(Rss)
1	0	0	0	1	0	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	0	1	0	d	d	d	d	d	Rd=cl0(Rss)
1	0	0	0	1	0	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=cl1(Rss)
1	0	0	0	1	0	0	0	0	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	0	0	0	d	d	d	d	d	Rd=normamt(Rss)
1	0	0	0	1	0	0	0	0	1	1	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	d	d	d	d	d	Rd=add(clb(Rss),#s6)
1	0	0	0	1	1	0	0	0	0	1	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	d	d	d	d	d	Rd=add(clb(Rs),#s6)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=clb(Rs)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	1	d	d	d	d	d	Rd=cl0(Rs)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	0	d	d	d	d	d	Rd=cl1(Rs)
1	0	0	0	1	1	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	1	d	d	d	d	d	Rd=normamt(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Count population

Population Count (popcount) counts the number of bits in Rss that are set.

Syntax	Behavior
`Rd=popcount(Rss)`	`Rd = count_ones(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=popcount(Rss)`	`Word32 Q6_R_popcount_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	0	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	0	1	1	d	d	d	d	d	Rd=popcount(Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Count trailing

Count trailing zeros (ct0) counts the number of consecutive zeros starting with the least significant bit.

Count trailing ones (ct1) counts the number of consecutive ones starting with the least significant bit.

Syntax	Behavior
`Rd=ct0(Rs)`	`Rd = count_leading_ones(~reverse_bits(Rs));`
`Rd=ct0(Rss)`	`Rd = count_leading_ones(~reverse_bits(Rss));`
`Rd=ct1(Rs)`	`Rd = count_leading_ones(reverse_bits(Rs));`
`Rd=ct1(Rss)`	`Rd = count_leading_ones(reverse_bits(Rss));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=ct0(Rs)`	`Word32 Q6_R_ct0_R(Word32 Rs)`
`Rd=ct0(Rss)`	`Word32 Q6_R_ct0_P(Word64 Rss)`
`Rd=ct1(Rs)`	`Word32 Q6_R_ct1_R(Word32 Rs)`
`Rd=ct1(Rss)`	`Word32 Q6_R_ct1_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType			MajOp				s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	1	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	0	1	0	d	d	d	d	d	Rd=ct0(Rss)
1	0	0	0	1	0	0	0	1	1	1	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=ct1(Rss)
1	0	0	0	1	1	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	0	d	d	d	d	d	Rd=ct0(Rs)
1	0	0	0	1	1	0	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	1	d	d	d	d	d	Rd=ct1(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

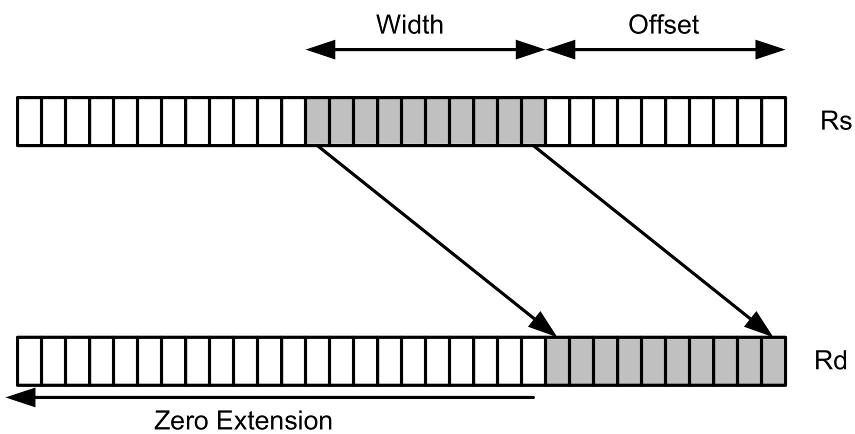
Extract bitfield

Extract a bitfield from the source register (or register pair) and deposit into the least significant bits of the destination register (or register pair). The other, more significant bits in the destination are either cleared or sign-extended, depending on the instruction.

The width of the extracted field is obtained from the first immediate or from the most-significant word of Rtt. The field offset is obtained from either the second immediate or from the least-significant word of Rtt.

For register-based extract, where Rtt supplies the offset and width, the offset value is treated as a signed 7-bit number. If this value is negative, the source register Rss is shifted left (the reverse direction). Width number of bits are then taken from the least-significant portion of this result.

If the shift amount and/or offset captures data beyond the most significant end of the input, these bits are taken as zero.



Syntax	Behavior

Rd=extract(Rs,#u5,#U5)	width=#u;
	offset=#U;
	Rd = sxt$_{width->32}$((Rs >> offset));

Rd=extract(Rs,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	Rd = sxt$_{width->64}$((offset>0)?(zxt$_{32->64}$(zxt$_{32->64}$(Rs))>>>offset):(zxt$_{32->64}$(zxt$_{32->64}$(Rs))<<offset));

Rd=extractu(Rs,#u5,#U5)	width=#u;
	offset=#U;
	Rd = zxt$_{width->32}$((Rs >> offset));

Rd=extractu(Rs,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	Rd = zxt$_{width->64}$((offset>0)?(zxt$_{32->64}$(zxt$_{32->64}$(Rs))>>>offset):(zxt$_{32->64}$(zxt$_{32->64}$(Rs))<<offset));

Rdd=extract(Rss,#u6,#U6)	width=#u;
	offset=#U;
	Rdd = sxt$_{width->64}$((Rss >> offset));

Syntax	Behavior

Rdd=extract(Rss,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	Rdd = sxt$_{width->64}$((offset>0)?(Rss>>>offset):(Rss<<offset));

Rdd=extractu(Rss,#u6,#U6)	width=#u;
	offset=#U;
	Rdd = zxt$_{width->64}$((Rss >> offset));

Rdd=extractu(Rss,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	Rdd = zxt$_{width->64}$((offset>0)?(Rss>>>offset):(Rss<<offset));

Class: XTYPE (slots 2,3)

Intrinsics

Rd=extract(Rs,#u5,#U5)	Word32 Q6_R_extract_RII(Word32 Rs, Word32 Iu5, Word32 IU5)
Rd=extract(Rs,Rtt)	Word32 Q6_R_extract_RP(Word32 Rs, Word64 Rtt)
Rd=extractu(Rs,#u5,#U5)	Word32 Q6_R_extractu_RII(Word32 Rs, Word32 Iu5, Word32 IU5)
Rd=extractu(Rs,Rtt)	Word32 Q6_R_extractu_RP(Word32 Rs, Word64 Rtt)
Rdd=extract(Rss,#u6,#U6)	Word64 Q6_P_extract_PII(Word64 Rss, Word32 Iu6, Word32 IU6)
Rdd=extract(Rss,Rtt)	Word64 Q6_P_extract_PP(Word64 Rss, Word64 Rtt)
Rdd=extractu(Rss,#u6,#U6)	Word64 Q6_P_extractu_PII(Word64 Rss, Word32 Iu6, Word32 IU6)
Rdd=extractu(Rss,Rtt)	Word64 Q6_P_extractu_PP(Word64 Rss, Word64 Rtt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	1	I	I	I	s	s	s	s	s	P	P	i	i	i	i	i	i	I	I	I	d	d	d	d	d	Rdd=extractu(Rss,#u6,#U6)
1	0	0	0	1	0	1	0	I	I	I	s	s	s	s	s	P	P	i	i	i	i	i	i	I	I	I	d	d	d	d	d	Rdd=extract(Rss,#u6,#U6)
1	0	0	0	1	1	0	1	0	I	I	s	s	s	s	s	P	P	0	i	i	i	i	i	I	I	I	d	d	d	d	d	Rd=extractu(Rs,#u5,#U5)
1	0	0	0	1	1	0	1	1	I	I	s	s	s	s	s	P	P	0	i	i	i	i	i	I	I	I	d	d	d	d	d	Rd=extract(Rs,#u5,#U5)
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rdd=extractu(Rss,Rtt)
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	d	d	d	d	d	Rdd=extract(Rss,Rtt)
1	1	0	0	1	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rd=extractu(Rs,Rtt)
1	1	0	0	1	0	0	1	0	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	d	d	d	d	d	Rd=extract(Rs,Rtt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type
`RegType`	Register Type